Proof reconstruction for first-order logic and set-theoretical constructions

Clément Hurlin

Proof reconstruction for first-order logic and set-theoretical constructions

Clément Hurlin [1]

LORIA and University of Nancy
BP 239, Campus scientifique
54506 Vandœuvre les Nancy, FRANCE

Abstract

Proof reconstruction is a technique that combines an interactive theorem prover and an automatic one in a sound way, so that users benefit of the expressiveness of the first tool and the automation of the latter. We present an implementation for proof reconstruction for first-order logic and set-theoretical constructions between Isabelle [11] and haRVey [3].

Keywords: proof reconstruction, set theory, first-order logic, harvey, Isabelle.

1 Introduction

Security and reliability are paramount for software development. While a variety of engineering practices have been developed to help increase quality (e.g., testing practices, system design, modern processes, robust operating systems and programming languages), the most promising way is the use of applied formal methods.

Formal methods, in particular verification of programs [7] or models [1], generate proof obligations. For example, to validate a program with respect to a specification or an invariant, users need to discharge proof obligations, and therefore need powerful deductive support tools. In this paper, we describe methods to discharge proof obligations for widely used languages: first-order logic and set-theoretical constructions.

First-order logic is at the basis of any deductive machinery. Formulas containing set symbols, such as

$$(X \cap Y = \emptyset) \wedge (X \setminus Z = X) \wedge (Y \cap Z \neq \emptyset) \longrightarrow X \cap (Y \cup Z) = \emptyset,$$

are encountered when using formal languages such as TLA+ [8] or B [1].

[1] Email: clement.hurlin@loria.fr

1.1 Expressiveness, automation and soundness

These three words resume the challenge for applied formal methods. Even if numerous well-known methods to decide the validity of different kinds of formulas exist, they do not often provide expressiveness, automation and soundness all together.

Automation enables people to use a prover without intimate knowledge of the tool's internals. Automatic tools, such as decision procedures, are highly optimized in order to handle large proof obligations. Yet, automation is only possible for languages of limited expressiveness. Furthermore, aggressive optimization increases the likelihood of bugs in the code. *Expressiveness* is provided by interactive theorem provers (e.g. [11,14,12]). These tools encode rich logics, which are very expressive and easy to extend, thus enabling a user to define appropriate abstractions for the problem at hand. Nevertheless, many actual proof obligations are quickly reduced to problems in limited fragments of logic, and this observation makes it interesting to combine automatic and interactive provers. *Soundness* is fundamental in deduction, and in particular for tool combinations. Many interactive tools sacrifice efficiency for soundness: provers in the *LCF* tradition such as Isabelle [11] reduce deduction to basic application of rules that can be implemented in a small, trusted kernel. Other provers such as Coq [14] achieve soundness by verifying proof terms afterwards.

1.2 Proof reconstruction

We describe the combination of the automatic prover haRVey [3] and the interactive proof assistant Isabelle. haRVey is an SMT [13] (Satisfiability Modulo Theories) prover. Isabelle is a well-known interactive prover which has been developed at the Technical University of Munich and at the University of Cambridge. Isabelle provides a backdoor for using trusted automatic tools as so-called *oracles*. Once the external tool has decided that a formula is a theorem, Isabelle will accept it as a theorem. However, this method may compromise the degree of soundness of Isabelle since a bug in the external tool would affect an Isabelle theorem. Moreover, even if one has a high degree of confidence in the external tool, the translation of the formula from Isabelle syntax to the external tool's syntax can easily introduce a bug.

Thus, our method to reach the three goals described above is *proof reconstruction* [15,5]. In a nutshell, the principle is to search for a proof within an automatic tool and then to replay it within the kernel of the interactive prover. Thus, we benefit of the automation of the first and the soundness of the latter; also, specifications can still be written in the expressive language provided by Isabelle. The proof search done by the automatic tool should not only result in a yes/no answer, but should also produce a proof trace, which contains enough information for the interactive tool to certify the result of the automatic prover. Certification should be automatic and deterministic, thus, from a user point of view, one just needs to push a button within the interactive tool that silently calls the automatic one and certifies the reasoning.

The automatic tool (haRVey) decides the satisfiability of a formula whereas the interactive one (Isabelle) decides its validity. We know that a formula is valid if and only if its negation is unsatisfiable. So, given a formula F in Isabelle, we send $\neg F$

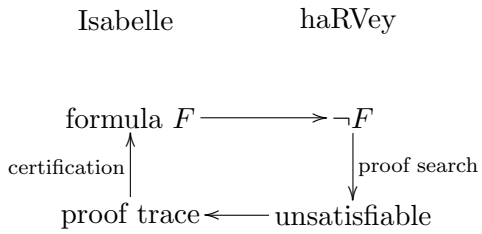

Fig. 1. Outline of proof reconstruction.

to haRVey, if haRVey returns that $\neg F$ is unsatisfiable then the proof reconstruction takes place. Otherwise, à priori F is not a theorem. The procedure is outlined in Fig. 1.

The proof trace is written in ML code, so that in Isabelle, we just need to read it on the fly (as Isabelle is written in ML). This is faster and more elegant than using a text file (which we would need to parse).

2 Implementing proof reconstruction

A technique for reconstructing proofs for formulas containing only uninterpreted function and predicate symbols plus equalities was described in [5], on which our work is based. We therefore only explain how we added proof reconstruction for quantified formulas.

Before describing the procedure for first-order logic, we explain how it is done for first-order logic without quantifiers having existential strength (a language we call ∀FOL). More precisely, the number of negations in front of a universal (respectively existential) quantifier must be even (respectively odd), considering that formulas of the form $\psi \longrightarrow \phi$ are transformed into $\neg\psi \vee \phi$.

2.1 First-order logic without quantifiers having existential strength (∀FOL)

Contrary to provers by resolution, haRVey handles formulas of the form $\forall x.P(x)$ by instantiating them in a brute-force manner. Briefly, we know that a set of formulas $\mathcal{F} \cup \{\forall x.P(x)\}$ is unsatisfiable if and only if there exists k instances $a_1, \ldots, a_k$ such that $\mathcal{F} \cup \{P(a_1) \wedge \cdots \wedge P(a_k)\}$ is unsatisfiable, where $a_1, \ldots, a_k$ belong to the Herbrand domain [4]. To replay that kind of reasoning within Isabelle, we have chosen to consider it slightly differently. Once haRVey has found the appropriate instances $a_1, \ldots, a_k$, it writes them in the proof trace. Then, in Isabelle, we add the implications $(\forall x.P(x)) \longrightarrow P(a_1)$, $\ldots$, $(\forall x.P(x)) \longrightarrow P(a_k)$ to the original formula. This results in an equivalent formula because each implication is valid. For example, this step transforms (1) into (2):

$$a \neq b \wedge \big((P \wedge \neg P) \vee \forall x.x = a\big) \tag{1}$$

$$\begin{aligned} &a \neq b \wedge \big((P \wedge \neg P) \vee \forall x.\, x = a\big) \\ &\wedge\ \underbrace{\forall x.(x = a) \longrightarrow a = a}_{\phi} \wedge \underbrace{\forall x.(x = a) \longrightarrow b = a}_{\psi} \end{aligned} \tag{2}$$

The conjunction $\phi \wedge \psi$ represents the fact that if $\forall x.x = a$ is true, so is $a = a$ and $b = a$. Of course, as first-order logic is undecidable, this method will not terminate in all cases, since instantiating $\forall x.P(x)$ can create an infinite conjunction of implications. Yet if the formula being checked is unsatisfiable (the case that interests us) and if the instances are selected in a fair way, this procedure will terminate.

2.2 First-order logic (FOL)

Compared to ∀FOL, full first-order logic (FOL) adds formulas of the form $\exists x.P(x)$. It is well known that such formulas are handled by skolemization. We say that a skolemization step on a formula F' ends in a formula F where a sub-formula of the form $\exists x.P(x)$ has been replaced by the formula $P(f(y_1, \ldots, y_n))$, where $x, y_1, \ldots, y_n$ are the free variables of $P(x)$ and where f is a new function symbol. By repeating this skolemization step, from a formula F' belonging to first-order logic, one can obtain a formula F belonging to ∀FOL, which is satisfiable if and only if F' is satisfiable (see [4,9] for more explanations).

Different techniques of skolemization exist, which differ in the arity of the new symbol introduced and/or in the degree of nesting of the new sub-formula. Yet, skolemizing a formula isn't proof search, it is a straight-forward transformation. That is why we found it pointless to reconstruct this step of the procedure. To extend our technique for first-order logic, it suffices to skolemize the formula before sending it to haRVey.

Thus, skolemization is implemented directly in Isabelle: first, all quantifiers are miniscoped[2], then the rule $\dfrac{\Gamma, \exists f.\forall x.P\ x\ (f\ x) \vdash \Delta}{\Gamma, \forall x.\exists y.P\ x\ y \vdash \Delta}$ is applied as many times as possible. Finally, existential quantifiers (which are all outermost at that point) are removed. After these transformations, the formula belongs to ∀FOL: we know how to decide its unsatisfiability and to reconstruct the reasoning.

2.3 Set-theoretical constructions (SET)

With a few restrictions, a formula containing set symbols (a language we call SET) can be transformed into an equivalent one which belongs to first-order logic (FOL). This is done by representing sets by their characteristic function. For any set S, we introduce a unary predicate $\breve{S}$ such that $\breve{S}(x)$ is true if and only if the element x belongs to the set S, and this inductively defines a translation of SET formula into FOL. For example, the SET formula $A \setminus \{a, b\} = B \cap C$ will be translated into the FOL formula $\forall x.\big[\big(\breve{A}(x) \wedge x \neq a \wedge x \neq b\big) = \big(\breve{B}(x) \wedge \breve{C}(x)\big)\big]$.[3]

Multiple ways to compute this transformation exist; haRVey implements it by rewriting set-theoretic symbols $(\subseteq, \subset, \in, \cap, \cup, \setminus)$ to lambda terms. It is easy to see that the resulting formula can be reduced to FOL by applying β-reduction and extensionality. In any case, this transformation isn't proof search, it is again just

[2] A quantifier is miniscoped when his scope is reduced. For example the formula $\forall x.P \wedge Q(x)$ is equivalent to $P \wedge \forall x.Q(x)$ because x is not a free variable of P (see [4] for detailed explanations).

[3] Note that this example uses extensionality (two sets A and B are equal if and only if $\forall x.\breve{A}(x) = \breve{B}(x)$), and that "=" is used to represent equality as well as equivalence.

computation. Like skolemization, there is no point to replay that kind of reasoning. That is why we implemented it directly in Isabelle.

Following the example of [2], we decided to implement the transformation from SET to FOL in two different ways: one classical tactic based, another one using the reflection principle (see [6]). Reflection was found to be two to ten times faster than the tactic-based translation. The performance can be further enhanced by using code extraction, as in [2,10]. [4]

3 Conclusion and further work

We have extended a technique for the sound integration of an automatic prover and an interactive proof assistant. Expressiveness is preserved because it can be used within the instantiation of Isabelle for higher-order logic, Isabelle/HOL. Automation is provided by haRVey, the Isabelle user just invokes a tactic that calls haRVey and performs proof reconstruction. Finally, proofs found this way come with a high degree of confidence because all steps are certified by the Isabelle kernel. We are not aware of other work having those qualities for languages as expressive as first-order logic and set-theoretical constructions. Furthermore, this work could be adapted to a wide range of other interactive and automatic tools.

On the implementation level, the format of the proof trace can be enhanced by writing Isabelle formulas as ML code instead of strings. This would reduce the running time, but requires some work on the haRVey side. Also, writing a generic interface dedicated to proof reconstruction can be useful to make cooperation between different tools easier.

Acknowledgement

I am grateful to Stephan Merz who supervised this work and revised this paper; to Pascal Fontaine for being my supervisor too, for his help with theory, haRVey implementation, for reviewing this paper and the valuable discussions we had; to Amine Chaieb for his great advices and to Tjark Weber who took me from level null to level *padawan* in the knowledge of Isabelle architecture.

References

[1] D. Cansell and D. Méry. Foundations of the B method. In *Computing and Informatics*, volume 22, pages 1–31, 2003.

[2] A. Chaieb. Mechanized quantifier elimination for linear real-arithmetic in Isabelle/HOL. Technical report, Technische Universität München, 2006.

[3] D. Déharbe, P. Fontaine, and S. Ranise. The haRVey theorem prover. http://harvey.loria.fr/.

[4] P. Fontaine. *Techniques for verification of concurrent systems with invariants*. PhD thesis, Institut Montefiore, Université de Liège, Belgium, September 2004.

[5] P. Fontaine, J-Y. Marion, S. Merz, L. Prensa Nieto, and A. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *TACAS 2006*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181, Vienna, Austria, 2006. Springer-Verlag.

[4] Note that, in this case, the soundness of the whole process relies on the soundness of the code generator.

[6] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.

[7] J.R. Kiniry and D.R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *CASSIS'2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128, January 2005.

[8] L. Lamport. *The book Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education, 2002.

[9] D. Loveland. Automated theorem proving: A logical basis. In *Fundamental studies in Computer Science*, 1978.

[10] T. Nipkow, G. Bauer, and P. Schultz. Flyspeck I: Tame graphs. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, 2006. To appear.

[11] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[12] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[13] S. Ranise and C. Tinelli. The smt-lib standard: Version 1.1. Technical report, 2005.

[14] The Coq development team. The Coq proof assistant reference manual v8.0. Technical Report 255, INRIA, France, March 2005.

[15] T. Weber. Integrating a SAT solver with an LCF-style theorem prover. In A. Armando and A. Cimatti, editors, *PDPAR'05*, Edinburgh, UK, July 2005.